Heurística: Proyecto 1
Facultad de Ciencias UNAM 2021-I

Ruiz Melo Jean Paul

Noviembre 13th

1 Introducción

El problema del agente viajero (or TSP) consiste en resolver la siguiente problema:

"Dado un subconjunto de ciudades en un mapa, ¿cuál es el camino más corto que pasa por todas ellas (si acaso existe)?"

Dado una gráfica G, con V el conjunto de todo las ciudades y E el conjunto de aristas con peso igual a la distancia entre ellos, queremos encontrar la distancia optima para recorre a todos una vez. Para este proyecto, si dos ciudades u y v no tienen una conexión, entonces le proponemos uno para ellos. Esta distancia le llamos la distancia natural y se define como:

$$\text{d(u, v)} = 6{,}373{,}000 \times 2 \times arctan(\sqrt{A}, \sqrt{1-A})$$

Donde A se define como:

$$\text{A} = \sin^2(\tfrac{lat(v)-lat(u)}{2}) + \cos(\text{lat(u)}) \times \cos(\text{lat(v)}) \times \sin^2(\tfrac{lon(v)-lon(u)}{2})$$

La distancia natural es entonces multiplicado por la arista de peso máximo que existe en G. Esto nos deja asegurar que todo solución que contiene una conexión entre dos ciudades que no existe tiene un peso mucho mayor a uno que esta en la gráfica. Para entonces encontrar un camino que si es posible existir definimos un función de costo. El función de costo toma la distancia entre cada ciudad en el camino i y i+1.

De esto, tambien calculamos un normal para saber que camino si es posible o no. Para hacer esto debemos dividir por un normal. Este normal lo podemos definir como la suma de las |S - 1| distancias más grandes en la gráfica, donde |S| es igual al número de ciudades en el camino. Con esto normal cuando dividimos el funcion de costo podemos saber lo siguiente:

- Si el resultado del funcion de costo es mayor que 1, el resultado no es factible ya que contine una arista con la distancia natural
- Si es menor que 0, entonces es factible. Lo mas cercano a 0 mejor es la respuesta

Definiendo un funcion de costo podemos empezar a encontrar una solucion usando la Heurística de Recocido Simulado. Mientras que encontrando la solucion más optima tardaria mucho, podemos acercarnos a ese solucion. Recocido Simulado se base en acercar a la solucion global minima usando un proceso de enfriamiento lento que es interpretado como un decaimiento lento en la probabilidad de aceptar una solucion peor en el espacio explorado. Aceptando soluciones malas propone un mejor busqueda de la solucion minima en el espacio explorado.

Para empezar tenemos que definir una temperatura. Una temperatura muy alto resula en tiempo perdido ya que la mejora de la solucion actual empieza a ser minuscula. Pero una temperatura muy baja puede resultar en atorarse en una minima local. Para encontrar una temperatura mejor hacemos una busqueda binaria donde nos basamos en aceptar una gran cantidad de soluciones, pero no todas.

La heurística propia es basado en lotes de soluciones. Dado un numero L que nosotros definimos, generamos soluciones nuevos hasta que encontramos L numero de soluciones que fueron mejores que el anterior. Seguimos generando lotes hasta que el promedio del función de costo de lotes generaros es mayor que el promedio del lote anterior. Bajamos la temperatura con enfriamiento y empezamos a generar lotes de nuevo hasta que la temperatura llega a un punto determinado por nosotros. Podemos garantizar que este proceso termina proponiendo un maximo de soluciones posibles de generar por Lote.

Finalmente, durante el proceso guardamos un minimo global, este minimo global es el solucion que se va a proporner al final.

2 Tecnologías usadas

- Lenguage de programacion: Originalmente este proyecto fue hecho en rust, pero debido a problemas con conectando al base de datos el lenguaje se cambio a Java por la comidad y costumbre a aquel.
- Compilador: ANT fue usado por el mismo razon dado anteriormente
- Imagenes: Gnuplot fue usado para generar las graficas

3 Design

Database.java
La informacion de los varios ciudades y conexiones estan guardados en un archivo .db. La clase Database le esos archivos y los guarda en los Objetos City, World y Connections.

City.java
Un City consiste de un numero de ciudad, el nombre de la ciudad, el país a que corresponde, su poblacion y so posición en latitud y longitud. Para este proyecto solo nos importa el numero de una ciudad, su longitud y latitud. El número de ciudad sirve como un identificado unico mientras que la longitud y latitud son importantes para calculando la distancia natural.

World.java
El World consiste de una lista de City's. La posicion de una ciudad en la lista corresponde a su numero menos uno, proporcionando el acceso immediato de una ciudad en la lista.

Connections.java
Connections consiste de una matriz de adyacencia de las distancias de las ciudades dado en el base de datos. Mientras que fue posible enfocar en solo un lado del diagonal de la matriz (ya que si una ciudad1 y ciudad2 estan conectados, entonces ciudad2 y ciudad1 tambien lo deben de ser), todo la matriz es llenando para convenencia. Cualquier dos ciudades que no estan conectados tienen su posición marcado con -1, ya que la distancia entre dos ciudades no pueden ser negativos.

Normalizer.java
El funcionamiento de Normalizer pueder ser definido con el siguiente:

Para cada par no-ordenado u,v ϵ S, si (u,v)ϵ E, entonces sumamos la distancia de w(u, v) a una lista L donde ordenamos a L de mayor a menor. L' es igual a L si el longitud de L es menor que |S|-1 o la sublista de L con su primer |S|-1 elementos en otro caso.

Podemos entonces tomar la suma de esta lista para obtener la distancia mas grande posible en la matriz de conneciones. Este 'normal' puede ser usado para decidir si la funcion de costo en la solucion actual es factible, ya que la distancia natural entre ciudades es mucho mas grande que cualquier distancia normal.

CostFunction.java
CostFunction es la clase que calcula que tan optimo es una solución. En otras palabras que tan cerca esta a un camino optimo y factible. Esto es hecho agregando la distancia de cada ciudad en el índice i con la ciudad en el índice i+1. Si no existe un camino entre esos dos ciudades, calculamos la distancia natural entre esos dos ciudades.

La suma de esta distancia es dividido por el normal, o en otras palabras, la distancia maxima posible en la gráfica que es hecho con el mismo número de ciudades que en la solucion actual. Usamos esto para saber si el camino de ciudades en la solucion actual es posible o no. Si es mayor que 1 este resultado, entonces existe al menos una pareja de ciudades que no tienen conexion. Si es menor que 1 entonces la solucion es factible.

Solutions.java
Solución consiste del camino actual de ciudades que esta siendo obersvado. Estas ciudades son guardados en un arreglo donde cada indice es igual al numero de ciudad. Cada solucion tambien contiene un World y Connections. Esto es para que cada solucion nuevo puede calcular su funcion de costo cuando es creado en constante. Pero tomando el tiempo revisar, puede resultar que crear un solucion nuevo puede estar tomando $O(n^2)$ si la copia de las conexiones se esta creando tambien. Además crear un nuevo solucion resulta en tener que crear un Deep Copy debido a como está implementado. Si no fuera por esto podria ser posible solo tomar O(1) en actualizar el funcion de costo.

Esto es debido a que una solucion nuevo solo necesita intercambiar dos ciudades diferentes. Entonces actualizar el funcion de costo solo tarda a lo más 8 calculos de suma y resta.

Temperature.java
Temperatura es una de las partes mas importante del Recocido Simulado. Si empieza a enfriar muy rapido la temperatura entonces es posible que no obtenemos una respuesta razonable, pero si enfría muy lento es posible que no termina el programa en un tiempo razonable, ya que el resultado obtenido en ese caso puede tener una mejor minúscula. Entonces se necesita encontrar una temperatura en un rango apropiado.

Calculando la temperatura incial es basado en los valores que nosotros damos, en particular el porcentaje de soluciones que queremos que regresan una solucion mejor que el anterior. La temperatura va subiendo o bajando hasta que ese porcentaje de aceptados se alcanza.

Heuristic.java
La heurística termina cuando la temperatura llega a un punto que nostros definimos anteriormente. La temperatura solo decrementa con base en lotes de soluciones. Estos lotes calcula un numero de soluciones hasta llegar a L soluciones que fueron mejores que el anterior. Estos regresan el promedio del funcion de costo de las soluciones aceptados. Como es posible que un mejor solucion no es encontrado, definimos tambien un numero maximo de soluciones que se pueden generar. Para este proyecto L es definido con L = (numero de ciudades)^2/5. Un mejor numero para L seria (numero de ciudades)^2/2 para que un poco mas de la mitad de posbile vecinos de soluciones son revisados. Definimos un vecino como un solucion

que difiere en solo dos ciudades.

Cuando el promedio regresado de un lote es mayor que el lote anterior, entonces la temperatura se enfria por una cantidad que nostros definimos. En este caso, solo 95% de la temperatura queda en cada iteracion. Mientras que va a resultar en una mayor tardanza, va a generar resultados mas completas. Durante todo este proceso, la solucion con el mejor funcion de costo va a estar guardado para poder regresar el mejor solucion encontrado.

Main.java
El main se encarga de leer los banderas que afecta el comportamiento del programa y la union de las clases mencionados anteriormente. Las banderas de comportan del siguiente forma:

- Seed: Deja que los acciones aleatorias como el swap de ciudades se pueden reproducir
- Gen: Genera los 5 mejores resultados de un numero n de semillas. Se puede iniciar el primer semilla con Seed, hasta n semillas despues item
- Path: Seleciona el archivo de que ciudades para el cual se quiere generar soluciones
- Out: Deja que los resultados se guardan en un archivo en lugar de ser imprimidos en un terminal

Despues de leer la bandera y la informacion del base de datos, la banderas definen el comportamiento del programa. Pero sin importar el numero de soluciones que se quieren generar, sigue el siguiente comportamiento:

- Encontrar el normal del solucio
- Encontrar la temperatura para la solucion
- Aplicar la heuristica
- Guardar los 5 mejores soluciones

4 Conclusions

Revisando el codigo al final de proyecto me deja creer que el mayor razon de porque tarda tanto es que a generar una solucion nuevo, World y Connections estan copiados tambien. Si eso es el caso, entonces en lugar de tarder n en generar una nueva solucion, tarda $O(n^2)$.

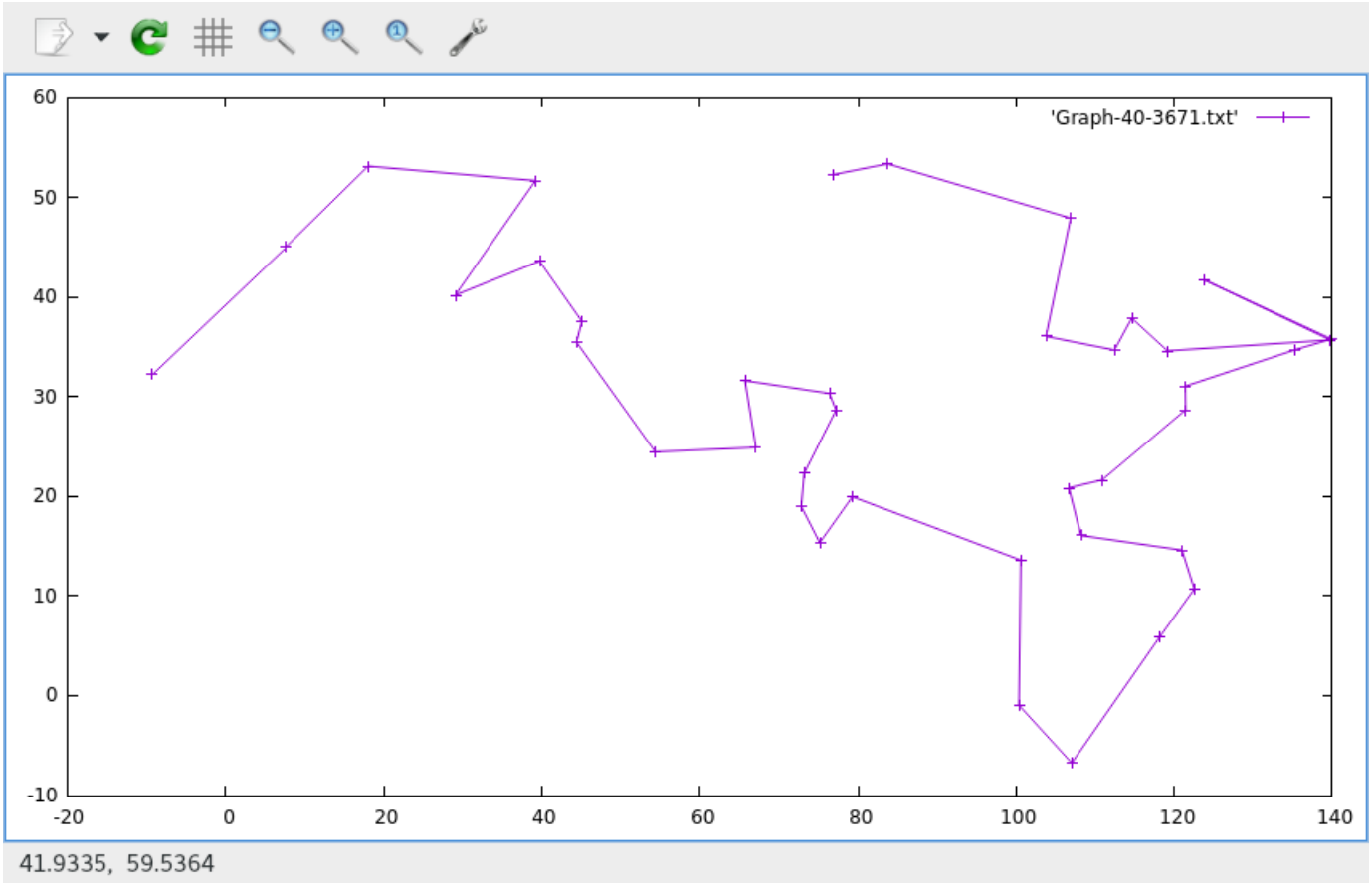
Los mejores soluciones para la instancia de 40 ciudades de 4000 semillas son los siguientes:

Semilla	Funcion de Costo
3671	0.21751778843543657
964	0.2208575645374767
1326	0.22141043636026847
139	0.22277951352860223
2412	0.22296760975517513
3377	0.22346558474361009
820	0.2239136372913415
3343	0.2240611819932075
2805	0.22537167705659103
3860	0.22579567351292754

El mejor resultado, la semilla 3671, resulta en el siguiente camino:

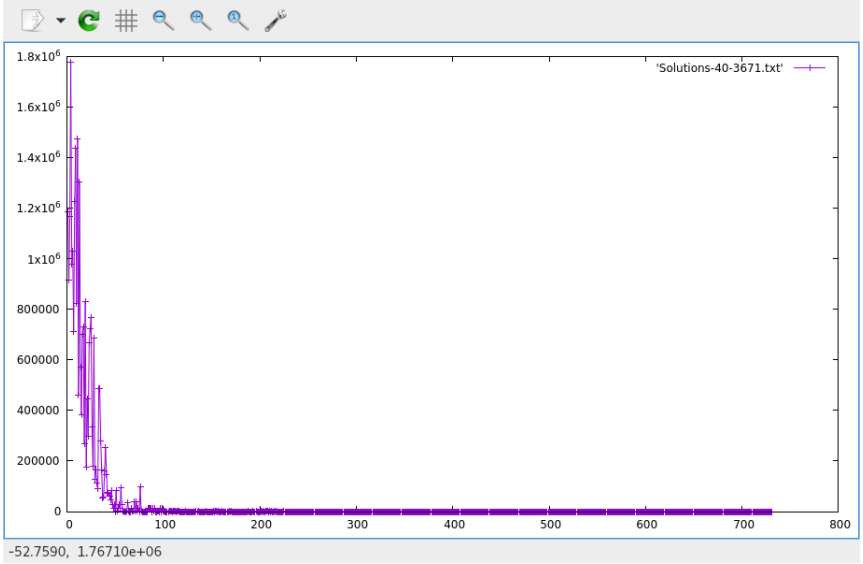
979,493,329,163,172,496,815,1,168,657,661,2,656,653,490,654,7,823,816,982,332,820,981,333,3,
165,6,5,978,817,4,489,492,491,984,164,331,871,327,980

Que puede ser representado con la siguiente grafica



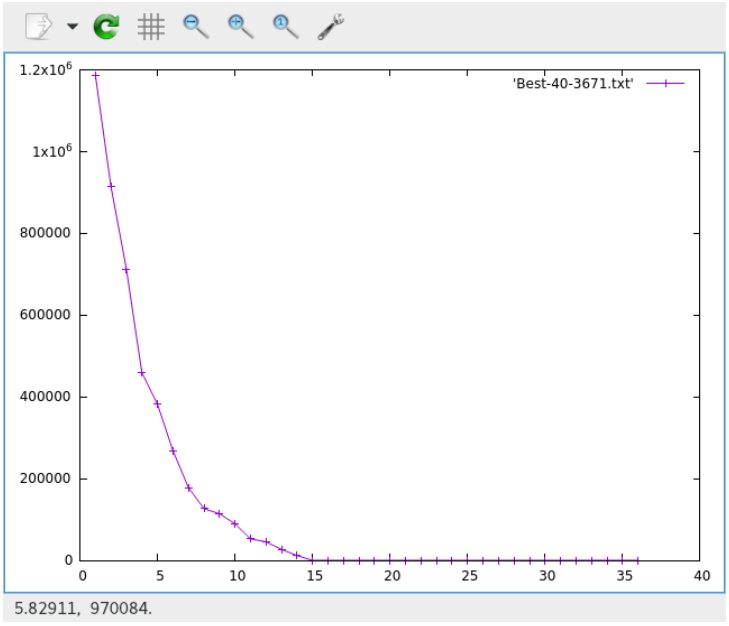
Podemos notar que hay casi ningun cruze en la grafica, lo que quiere decir que esta posiblemente cerca a una mejor solucion.

Podemos ver el progreso de la generacion de la solucion en la siguiente grafica:



Lo que me resalta es que una solucion factible es obtenido relativamente rapido pero la mejora se puede ver que es muy poco despues. Para gastar menos tiempo el epsilon que determina cuando termina o el enfiramiento podrian ser ajustado para gastar menos tiempo.

La siguiente grafica solo guarda los minimos globales, pero podemos ver que el numero de mejoras son muy pocas en total.



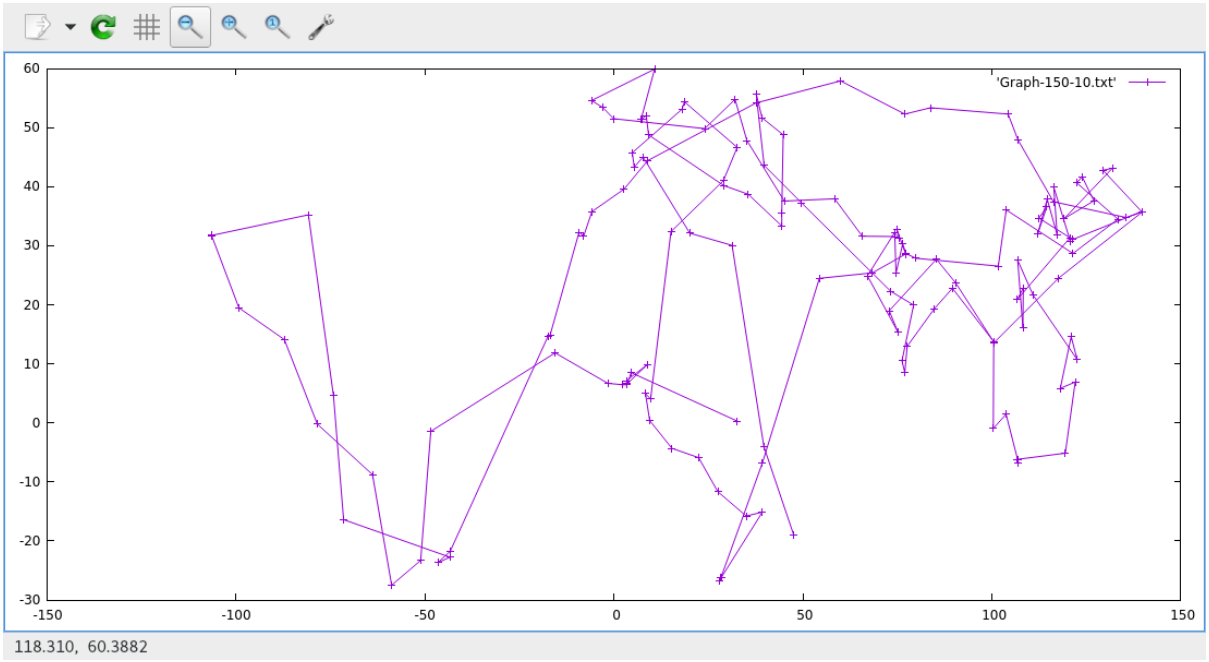
Los 5 mejores soluciones de la instancia de 150 de 50 semillas son los siguientes:

Semilla	Funcion de Costo
10	0.22563770969196262
14	0.2271077841637491
46	0.22743108693616382
9	0.0.229985277375067
12	0.2308175880901926

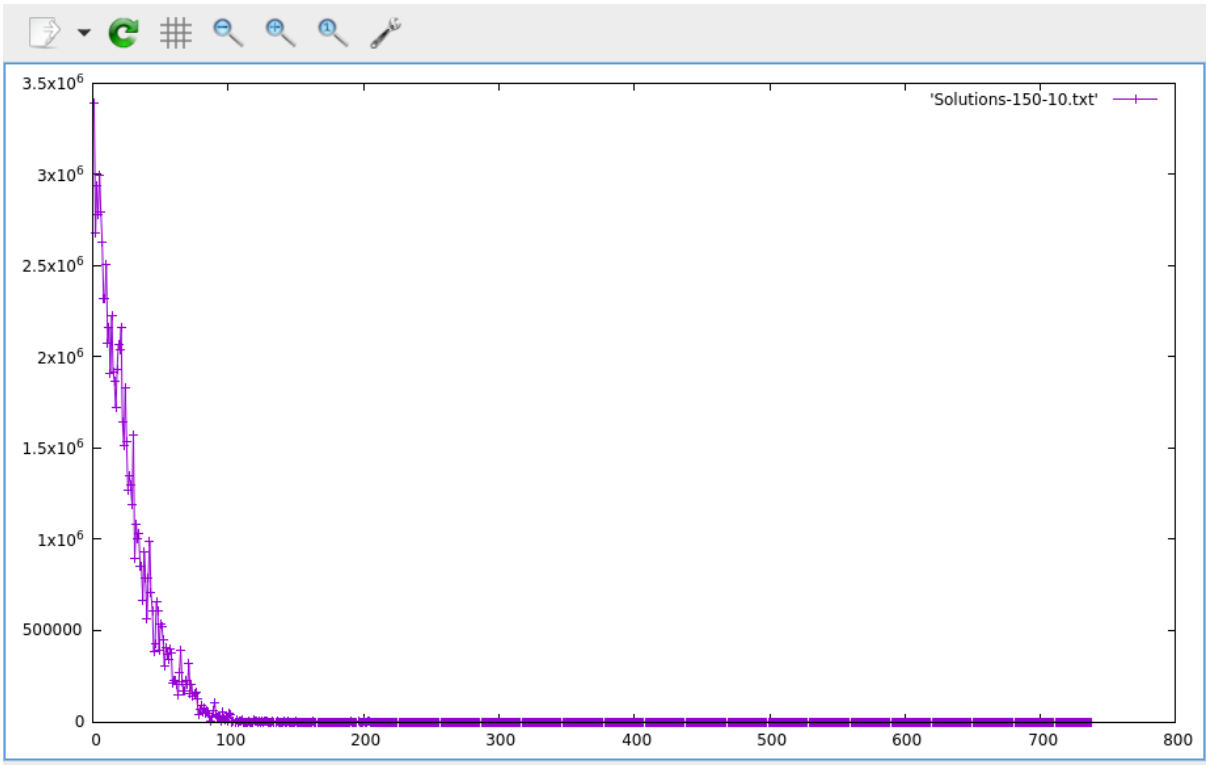
El mejor semilla resulta en el siguiente camino:

183,340,1038,339,502,12,828,151,822,166,655,819,1073,169,326,328,495,167,494,1037,330,666,818,658,74,1001,
980,297,336,840,511,662,837,979,493,509,329,839,829,661,657,663,2,172,182,673,496,173,19,667,184,490,344,
654,665,653,823,7,816,678,187,14,181,982,345,332,26,185,991,27,353,990,981,165,499,984,8,331,995,492,25,501,
164,504,985,500,825,343,510,660,20,334,999,349,491,347,817,23,988,978,5,1004,676,163,656,832,505,168,9,815,
508,986,1,507,820,22,351,3,333,4,176,352,668,6,174,489,75,483,652,1075,821,512,179,671,16,520,675,186,826,
11,1003,674,871,670,350,327,444,17,346,171

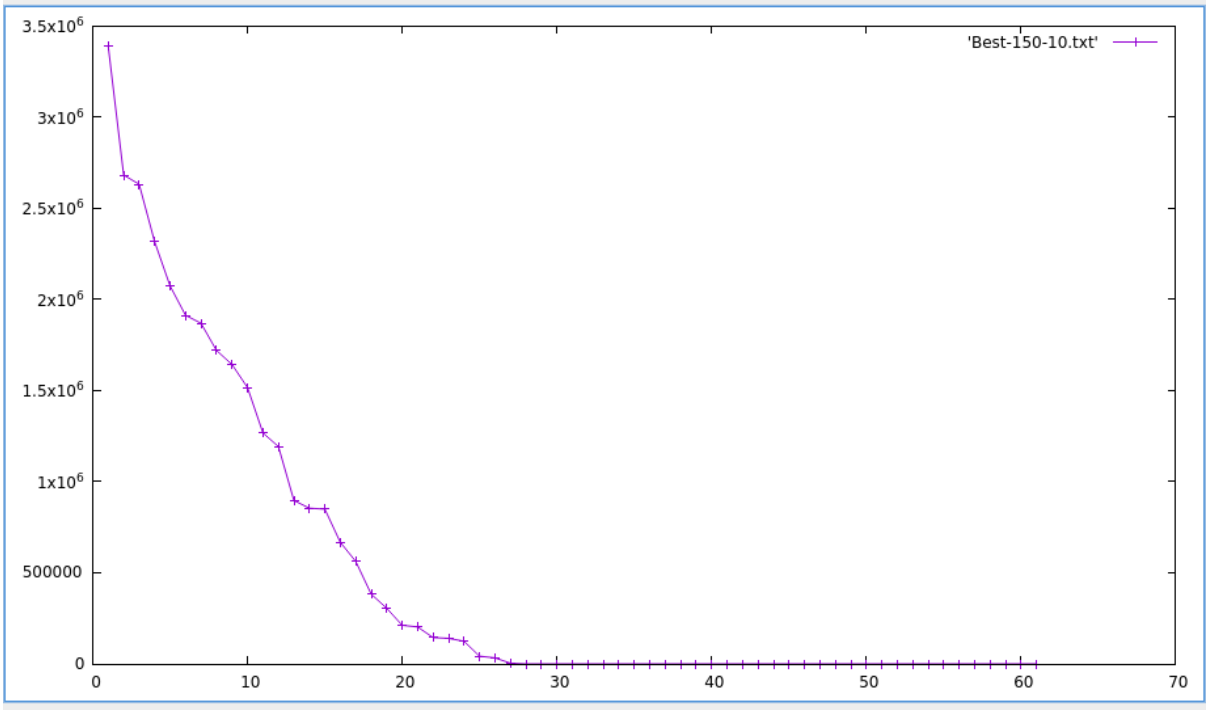
Que resulta en la siguiente grafica:



Lo que resulta como obvio es que hay demasiados curzes y interseciiones. Lo que creo es que necesita mas tiempo para una solucion optima para ser encontrado. Pero no en la temperatura si no en los lotes, ya que en las siguientes graficas podemos notar un comportamiento similar a la instancia de 40. Como L solo revisa un numero bajo de posible vecinos, incrementar el numero de vecinos revisados podria mejorar la calidad de cada lote.



205.712, 2.96878e+06



13.5104, 3.42884e+06

Mejorar la programa se podria ver en ajustar que la temperatura termina mas rapido y que el numero de lotes se incrementa. Si se resolve el tiempo de generar una nueva solucion, incrementar L no deberia afectar el tiempo de ejecuccion tanto. Como esta el programa, mas grande que esta la instancia pero va a funcionar.

5 Bibliography

https://www.quantamagazine.org/computer-scientists-break-traveling-salesperson-record-20201008/
https://en.wikipedia.org/wiki/Travelling_salesman_problem
https://en.wikipedia.org/wiki/Simulated_annealing